Tender: Accelerating Large Language Models via Tensor Decomposition and Runtime Requantization

Jungi Lee* Wonbeom Lee* Jaewoong Sim
Department of Electrical and Computer Engineering
Seoul National University
{jungi.lee, wonbeom, jaewoong}@snu.ac.kr

***Abstract*—Large language models (LLMs) demonstrate outstanding performance in various tasks in machine learning and have thus become one of the most important workloads in today's computing landscape. However, deploying LLM inference poses challenges due to the high compute and memory requirements stemming from the enormous model size and the difficulty of running it in the integer pipelines. In this paper, we present Tender, an algorithm-hardware co-design solution that enables efficient deployment of LLM inference at low precision. Based on our analysis of outlier values in LLMs, we propose a decomposed quantization technique in which the scale factors of decomposed matrices are powers of two apart. The proposed scheme allows us to avoid explicit requantization (i.e., dequantization/quantization) when accumulating the partial sums from the decomposed matrices, with a minimal extension to the commodity tensor compute hardware. Our evaluation shows that Tender achieves higher accuracy and inference performance compared to the state-of-the-art methods while also being significantly less intrusive to the existing accelerators.**

***Index Terms*—Large Language Model; LLM Acceleration**

I. Introduction

Large language models (LLMs) have demonstrated remarkable performance across a variety of tasks in natural language processing, including machine translation, sentiment analysis, and even generating human-like text, as evidenced by recent applications such as OpenAI's ChatGPT and Google's Gemini [1], [19], [42], [56], [64]. The tremendous success of LLMs can be largely attributed to their enormous model size, which has seen substantial growth in recent years. For example, the first version of GPT, which was introduced in 2018, had 117 million parameters, but the recently released GPT-4 is rumored to contain more than a trillion parameters only after five years.

LLM inference has now become one of the most important workloads in today's computing landscape, but deploying and serving LLMs poses a unique challenge because it requires a significant amount of compute and memory resources due to the massive model size. Quantization [11], [18], [50], [66] is one of the most popular techniques to mitigate the resource problem. By quantizing *both* weights and activations in LLMs into low-bit integers, we can accelerate compute-intensive operations such as matrix multiplication while leveraging high-throughput integer tensor compute units in modern GPUs or TPUs, in addition to benefiting from memory capacity and bandwidth savings.

However, it is quite challenging to quantize the activations in LLMs, unlike convolutional neural networks or small Transformer models. When the LLM scales beyond a certain size (around 6.7B parameters), extremely large magnitude values, compared to others, appear in a few feature dimensions of activations [11]. These *outliers* increase the quantization range, thereby necessitating the use of larger bit widths in LLMs compared to other DNN models.

As such, there have been recent efforts to effectively quantize activations in LLMs using low-bit integers in both software and algorithm-hardware co-design works. However, most software-only works do not noticeably reduce the inference time due to the overhead of complex algorithms or result in a significant quantization loss at ultra low-bit precisions (e.g., 4 bits) [11], [33], [65], [66]. Also, prior works dealing with outliers via algorithm-hardware co-designs require either mixed-precision/complex compute units [11], [44], [52], [70] or custom/adaptive datatypes that are not natively supported by commodity hardware [20], [21], [55].

In this paper, we propose Tender, an algorithm-hardware co-design technique that efficiently executes LLMs in the high-throughput integer pipeline without the need of mixed-precision compute units or custom/adaptive datatypes. The high-level idea behind Tender is to split the *activation* tensor into several *subtensors* along the feature/channel dimensions (e.g., columns in 2D), each of which contains the elements with a similar range of values, effectively isolating the channels that contain outliers from the others. Each subtensor is then quantized with a different scale factor, thereby reducing the quantization error for the entire activation tensor compared to the conventional per-tensor (or per-row) quantization variants.

While this has the great potential to enhance model performance compared to previous LLM quantization works, setting different scale factors for each subtensor requires *explicit* and *costly* rescaling/requantization (i.e., dequantization/quantization) when adding up the partial sums (i.e., outer products) from matrix multiplications of the subtensors. Our key insight is that we can avoid the explicit requantization step by setting the scale factors with *power-of-two* relationships between the decomposed subtensors and employing simple

*Equal contribution

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00080

shifter logic in the tensor compute units. This approach provides us with two key benefits. First, it makes requantization *implicit*, being it performed along with matrix multiplication without involving explicit floating-point operations, leading to negligible overhead for rescaling when accumulating the partial sums. Second, it enables higher model performance while being significantly less intrusive to the conventional tensor compute units as it does not require complex hardware to handle mixed-precision or custom datatypes, thereby offering more flexible and practical applicability than other algorithm-oriented schemes or outlier-aware architectures. While the Tender algorithm can be implemented in software, its full potential is realized through a custom accelerator design that supports implicit requantization, which we discuss in Section IV.

We apply our decomposed quantization technique to three representative LLMs for evaluation. To measure the performance improvement, we implement a cycle-level simulator that models the Tender hardware with a detailed off-chip memory timing model. We use simulation parameters based on our RTL implementation, which is synthesized with a 28 nm process node. Our evaluation shows that INT8 quantization via Tender offers better model performance than the state-of-the-art and retains comparable model performance to the FP16 baseline (e.g., less than a 0.1 increase in perplexity for OPT-6.7B, OPT-13B, and OPT-66B). In INT4 quantization, Tender outperforms any other outlier-aware post-training quantization (PTQ) techniques (up to 10988× lower perplexity). Also, the Tender hardware achieves up to an average of 2.63× speedup over the outlier-aware accelerators that we evaluate. In summary, this paper makes the following contributions:

- We propose Tender, a PTQ approach for LLMs in pursuit of hardware performance as well as model accuracy. Tender achieves high performance and accuracy without the need of mixed-precision compute units or custom datatypes even for low-bit quantization.
- We propose the "power of 2" channel decomposition rule, which effectively reduces the quantization error by harmonically working with LLM activation tensors.
- We present the Tender accelerator design, which overcomes performance challenges in splitting activations along the channels and enables implicit/runtime requantization with negligible rescaling overhead, with a minimal extension to the commodity tensor compute hardware.

II. Background and Motivation

A. Large Language Models

In essence, LLMs build on a series of Transformer blocks [59], each of which consists of the *attention* and *feed-forward* layers, as illustrated in Figure 1. In this section, for ease of understanding the subsequent sections, we briefly describe the operation flow within the Transformer block using the terminology and notation used throughout the paper.

In the attention layer, there are four weight matrices, which are used for linear projections: query (W_Q), key (W_K), value (W_V), and output (W_O) projection matrices. Each projection matrix has a dimension of $d_{model} \times d_{model}$, where d_{model} is the dimension of an input embedding vector (i.e., the number of features). For an input sequence length of n (i.e., n tokens), the input (X) to the attention layer is an $n \times d_{model}$ matrix. Then, each of the query (X_Q), key (X_K), value (X_V) matrices of the attention layer is computed by multiplying the input matrix (X) with their respective weight matrices (W).

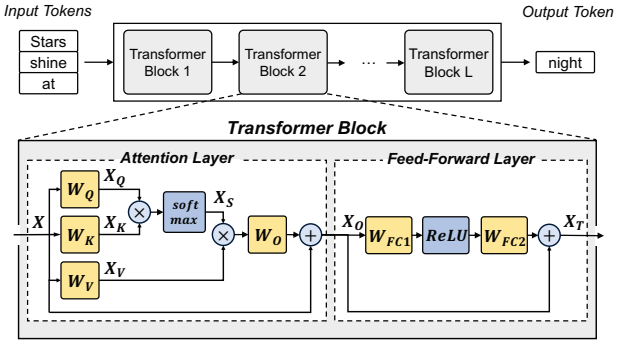


Figure 1: Illustration of the Transformer block architecture.

$$X_Q = X \times W_Q; \quad X_K = X \times W_K; \quad X_V = X \times W_V$$

After the QKV projection, we compute the attention score (X_S) and the attention value ($X_S \times X_V$) using the projected matrices. Lastly, the output of the attention layer (X_O) is computed by multiplying the attention value ($X_S \times X_V$) with the output weight matrix (W_O) along with a residual add (X).

$$\begin{aligned} X_S &= \quad \text{softmax}(X_Q \times X_K^T) \\ X_O &= \quad X_S \times X_V \times W_O + X \end{aligned}$$

The feed-forward network (FFN) in the Transformer block takes as input the output of the attention layer (X_O). It consists of two fully-connected (FC) layers, thus there are two weight matrices, W_{FC1} and W_{FC2}, each of which has $d_{model} \times h$ and $h \times d_{model}$ dimensions. The output of the feed-forward layer (X_T) is computed using the following equation.

$$X_T = \quad \text{ReLU}(X_O \times W_{FC1}) \times W_{FC2} + X_O$$

In addition, there exists a layer normalization operation (LayerNorm) at the start or the end of each attention and FFN layer, which we omit in the figure for simplicity. The Transformer block produces an output with the same dimensionality as the input, which makes Transformer-based LLMs easily scalable by adjusting the number of Transformer blocks.

B. Outliers in Large Language Models

The state-of-the-art post-training quantization (PTQ) methods [16], [33] demonstrate that the weights in LLMs can be effectively quantized to eight or even four bits via standard per-tensor (e.g., a scaling factor for each tensor) or grouping-based (e.g., a scaling factor for g consecutive weights in a tensor) quantization techniques without a significant degradation in model performance. In contrast, it is quite challenging to quantize *activations* in LLMs using the same methods due

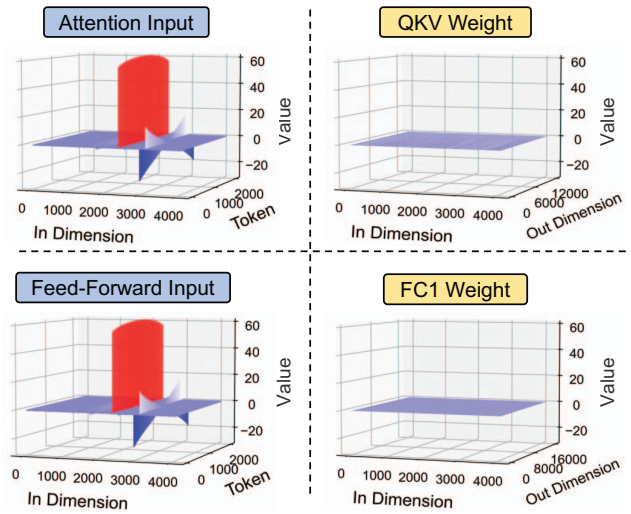


Figure 2: Values in the activation (left) and weight (right) tensors for the attention and the first FC layers. The values are obtained from the 8th layer in the OPT-6.7B model. The Q, K, V weight tensors are concatenated along the out dimension for better visualization.

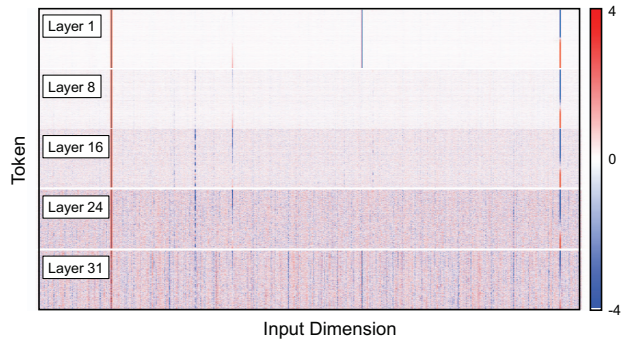


Figure 3: Heatmap of the attention input tensors for sampled layers in the OPT-6.7B model. The values larger than 4.0 or smaller than -4.0 are truncated to 4.0 or -4.0 for better visualization. We also only show channels from 2300 to 3000 for a clearer view.

to the existence of *outliers*, which refer to the extremely large magnitude values compared to others within a tensor.

Figure 2 shows the values in several weight and activation tensors. As depicted in the figure, the input activation tensors of the attention and the first FC layers (i.e., X and X_O) have significantly large values in a few input (feature) dimensions, whereas the weight tensors have a relatively similar range of values. In principle, this makes it difficult to quantize activations compared to the weight tensors in LLMs due to the wide range of values, which we discuss further in the following sections. Prior studies show that outliers are concentrated in the fixed channels of activation tensors across the layers and batches [11], [62] due to the model intrinsic, such as large LayerNorm weights in the fixed channels across the layers. We also observe a similar trend, as shown in Figure 3. We see the presence of vertical red or blue lines for each attention input tensor. This indicates that the outliers are typically at the channel granularity (i.e., within a few specific channels).

TABLE I: Model performance (perplexity) at different quantization granularities for activation tensors. Lower is better.

Models	OPT-6.7B	OPT-13B	Llama-2-7B	Llama-2-13B
FP16	10.86	10.13	5.47	4.88
INT8 per-tensor	26.73	4E+3	8.54	51.45
INT8 per-row	20.02	3E+3	5.58	4.94
INT8 per-column	**10.87**	**10.13**	**5.48**	**4.89**
INT4 per-tensor	1E+6	9E+8	4E+4	2E+4
INT4 per-row	1E+6	1E+9	1E+3	5E+3
INT4 per-column	**19.38**	**14.60**	**7.73**	**6.47**

C. Challenges in Quantizing Large Language Models

Preliminaries on Quantization. There have been various quantization techniques proposed over the past years. However, the most commonly used one today is *uniform integer* quantization, as it is amenable to acceleration by the integer pipeline in commodity hardware. As such, similar to prior works in LLM quantization [11], [65], this section focuses on uniform symmetric quantization, which can be expressed as follows:

$$s = \frac{x_{max}}{2^{b-1}-1}; \quad x_q = round(\frac{x_f}{s}),$$

where s is a scale factor, b is the bit width of a quantized value, x_{max} is the absolute maximum value, and x_f and x_q are a floating-point value and the quantized one. Dequantization restores quantized integer values to floating-point numbers by multiplying them with s. To mitigate the overhead of determining x_{max} in activation during runtime, most prior works employ *static quantization*, which pre-computes the scale factors using some calibration samples before runtime.

Depending on how elements in a tensor are being quantized as a group, there can be various levels of quantization granularity, including per-tensor, per-row, and per-column quantization. In per-tensor quantization, all the elements within the tensor share the same quantization parameter (i.e., a scale factor), which simplifies the quantization process. Per-row or per-column quantization shares the parameter at a row or column granularity to further reduce the quantization error.

Table I shows the perplexity when we quantize the activation at three different granularities. As shown in the table, the per-column granularity (i.e., input/feature dimension) shows the best perplexity, as it uses the quantization parameters for outliers that differ from others. However, applying per-column quantization to activations poses challenges in the modern GPU or TPU integer pipelines since each element needs scaling during the reduction operations in matrix multiplication. Consequently, all prior LLM quantization works employ per-row (per-token) or per-tensor quantization for activations and per-column or per-tensor quantization for weights.

Approaches to Handling Outliers in LLM Activations. Several algorithm-oriented PTQ works aim to handle outliers in LLM activations, and two most closely related works are LLM.int8() [11] and SmoothQuant [65]. LLM.int8() [11] employs a mixed-precision decomposition, where outliers in activations are kept in FP16 precision, while the other values are quantized to INT8. However, the mixed-precision decomposition leads to a non-negligible performance overhead

in performing matrix multiplication due to the dequantization that involves floating-point operations (Section III-B). SmoothQuant [65] addresses the quantization difficulty by partially migrating it from the activations to weights. However, there exist inefficiencies because it does not explicitly isolate outliers from normal values, leading to a large quantization loss at ultra low-bit precisions (Section V-B).

There are also several outlier-aware accelerators that use mixed precision to quantize normal values into low bits while separately handling the outliers. GOBO [70] is a weight-only quantization scheme that quantizes outlier weights using higher precision. OLAccel [44] uses a few 16-bit MAC units alongside 4-bit MAC units to deal with outliers. DRQ [52] employs a fine-grained detection algorithm to identify sensitive regions in a tensor. All of these works use mixed precision, requiring complex hardware and unaligned memory access. OliVe [20] is the most recent work, which quantizes outliers using custom number representations. Although there is no mixed precision involved, it prunes adjacent normal values and requires an encoder/decoder to support its custom datatype.

D. Challenges and Opportunities

Intuitively, one can choose to split activation tensors along the reduction axis (i.e., columns in 2D) and quantize each group with different scaling factors instead of employing an impracticable per-column approach. We can then represent the matrix multiplication via the channel grouping as follows:

$$P_i = \frac{X_i \times W_i}{s_i s_w}, \quad Y = \sum_{i=1}^{G} (s_i s_w) \cdot P_i, \tag{1}$$

where s_i and s_w are the scale factors of activation of group i and weight. P_i, Y, and G denote the partial sum from group i, the final resulting matrix, and the number of groups.

The above execution model still suffers from lower utilization of compute cores due to smaller submatrices and frequent rescaling to accumulate partial product P_i. Thus, to fully benefit from quantization while preserving the accuracy of the model, we need to address the following challenge—splitting channels along the reduction axis and grouping channels with similar ranges to isolate outliers from the others while retaining the reduction axis to better utilize compute cores.

Our intuition is that we can retain the reduction axis of matrix multiplication by processing partial sums in a specific order and rescaling the accumulated value before adding the next partial sum. Our execution model can be expressed as the following equation:

$$\begin{gathered} A_1 = P_1, \quad A_{i+1} = A_i \cdot \frac{s_i}{s_{i+1}} + P_{i+1}, \\ Y = A_G \cdot (s_w s_G), \end{gathered} \tag{2}$$

where $\mathrm{A_i}$ represents the results accumulated up to ith partial sums. Note that Equations 1 and 2 are mathematically equivalent. For Equation 2 to operate efficiently in hardware, the rescaling needs to be performed by the integer MAC units while preserving correctness. We will use the term *rescale factor* to denote s_i/s_{i+1} between the channel groups.

To this end, we propose Tender, an algorithm hardware co-design technique for quantizing large language models entirely into INT4/INT8 without using mixed precision, custom datatypes, or re-training. The carefully designed post-training quantization (PTQ) algorithm of Tender decomposes channels to isolate outlier channels from normal ones in activation tensors. At the same time, Tender enables rescaling inside the tensor compute units, so it *does not* require explicit requantization and thus fully utilizes the integer pipelines. In the following sections, we discuss how the above rescaling can be done inside the compute units with minimal extensions to hardware through the software-hardware co-design of Tender.

III. Algorithmic Implementation

A. Overview

In this section, we introduce tensor decomposition and runtime requantization of Tender. As discussed in Section II-B, outliers reside along the channels in the activation tensors across the layers of LLMs. Thus, we decompose channels to minimize the quantization error by isolating the outlier channels from the others. We propose a "Power of 2" channel decomposition rule that aligns well with the value distribution of activation tensors. Also, we present runtime requantization that works harmonically with the decomposition rule. It enables matrix multiplication between decomposed, quantized activation tensors and linear symmetrically quantized weights without involving floating-point operations but in a mathematically equivalent way. We then present a walking example of the Tender algorithm and optimizations at the end of the section.

While the explanation in this section is based on INT8 quantized activation-weight matrix multiplication, the same algorithm is applied to INT4 or activation-activation matrix multiplication (e.g., $X_Q \times X_K^T$ in a Transformer block). Furthermore, Tender can be easily extended to other bit widths (e.g., 5, 6, 7-bit integers) in the same way if the hardware supports such datatype operations. This is possible due to the standard and symmetric quantization of Tender, whereas other approaches typically need to define new custom datatypes.

B. Tender: Decomposed Quantization

Tender Computation Flow. As mentioned in Section III-A, we decompose activation tensors throughout the attention and feed-forward layers for quantization. Figure 4 shows our decomposed quantization strategy. First, we compute the bias of each channel and subtract it from the activation tensor. The bias is a similar concept to the zero-point used in asymmetric quantization, and it is computed as the sum of the maximum and minimum values divided by two. By subtracting the bias, Tender ensures that the absolute values of the maximum and minimum elements in the channel are equal, thus optimizing the bit usage.

Then, to accommodate the presence of outliers in quantization, we decompose the channels of the activation tensor into multiple groups of subtensors and use separate scaling factors for each group. Through runtime requantization, we multiply

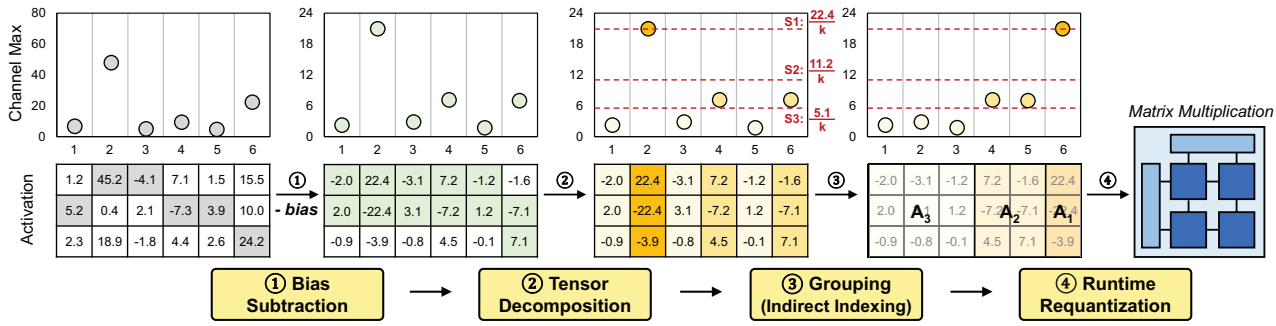


Figure 4: Decomposed quantization flow in Tender.

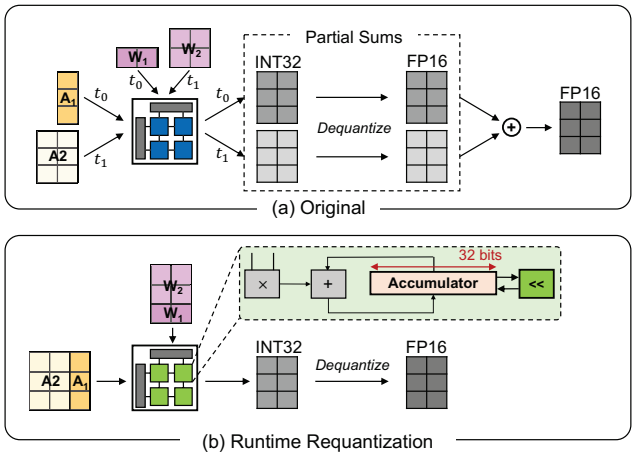


Figure 5: Runtime Requantization. Compared to the original computation flow, we retain the reduction axis length by shifting the accumulated partial product sum between decomposed matrices.

the decomposed quantized activation tensors and the linear symmetrically quantized weight without explicit dequantization with negligible latency overhead. Finally, the bias multiplied with weights is added to the output for mathematical correctness. Note that all the weights can be quantized to INT8 before inference. Also, channel decomposition, channel biases, and scale factors are pre-computed during calibration.

Tensor Decomposition. Tender decomposes channels by setting thresholds that are in powers of integer relationships. Decomposition consists of three steps. First, Tender finds the absolute maximum values of each channel (CMax). Tender also computes the maximum value from CMax, which corresponds to the absolute maximum of a given tensor (TMax). Then, we set the boundaries for splitting channels by dividing TMax with the power of α and assign i-th channel to an appropriate group g satisfying the following equation:

$$\frac{\mathrm{TMax}}{\alpha^{\mathrm{g}}} < \mathrm{CMax_i} \leq \frac{\mathrm{TMax}}{\alpha^{\mathrm{g}-1}}, \quad g = 1, 2, ...G \tag{3}$$

where $\mathrm{CMax_i}$ is the i-th channel absolute maximum. This is simple classification, which is much faster than clustering. Finally, every channel in group g is quantized using the same scale factor $\frac{\mathrm{TMax}}{\alpha^{g-1} \cdot (2^{b-1}-1)}$. If we choose an integer for α and compute channel groups with an ascending group index, rescaling can be done using integer arithmetic by simply multiplying α to an accumulated value. We use 2 for α, so that the requantization can be done with simple shifting. More details about the reason why we use the power of α approach and 2 for α will be discussed in the following section.

Power of 2. Here we explain our "power of 2" approach by answering the following three questions: 1) Why use classification? 2) Why use coarse-grained thresholds for large values and fine-grained thresholds for small values? 3) Why use 2?

1) Why use classification?: We can decompose channels in various ways. In the extreme scenario, we can use individual scale factors for each channel, which can greatly reduce the quantization error [5], [65]. However, in this way, as Figure 5(a) shows, the number of operations including matrix multiplication, dequantization, and addition between partial products increases in proportion to the number of channels. In the case of OPT-6.7B [72], for example, it increases 4096 times compared to the original computation without channel decomposition. As such, grouping channels that have a similar range of values to share a scale factor via clustering or classification is a more practical and scalable option. Clustering computes how similar the value range is between every channel and then groups the channels with a small distance. Classification is based on pre-defined thresholds where the channels are binned to fit within the threshold range. Thus, clustering can group the channels more accurately than classification but is not likely applicable at runtime (without complex hardware) due to the large computational overhead. So, we choose to use classification to make our algorithm simple and to be easily extended to support runtime decomposition.

2) Why use coarse-grained thresholds for large values and fine-grained thresholds for small values?: The maximum quantization error is $0.5 \times (\text{scale factor})$ since the maximum value of rounding error is 0.5. The scale factor is set proportional to the absolute maximum of the group. So, as the absolute maximum of the group becomes larger, the quantization error grows linearly. Also, considering that there exist multiple channels in a group, the quantization error of a group grows linearly to the number of channels that are included in the group. We can express the quantization error of a group as follows:

$$Q_{\mathrm{err}} \propto \text{Absolute Maximum} \times \text{Number of Channels}$$

Thus, for a group with a large absolute maximum, we need to minimize the number of channels in the group. Intuitively, this can be viewed as isolating quantization errors due to the large scale factor to only a few number of channels. Similarly, for a group with a large number of channels, we need to minimize the absolute maximum of the group. As Figure 2 shows, only a few channels have large magnitude values, and most of the channels have values near zero in the activation tensor. Thus, using coarse-grained thresholds (i.e., a large absolute maximum) for the channels with large magnitude values does not hurt the overall accuracy because the group includes only a few channels. Meanwhile, using fine-grained thresholds (i.e., a small absolute maximum) for the channels with small magnitude values can effectively minimize the accuracy drop by using the small threshold near the value range of the channels. Thus, our approach can efficiently classify the channels with a minimum accuracy drop. Note that, of course, it is possible to achieve better accuracy by using fine-grained thresholds for both the channels with large magnitude values and channels with small magnitude values. However, dividing more channels incurs a computation overhead, whereas using a fine-grained threshold for the channels with large magnitude values has a minimal increase in accuracy due to the small number of channels that are included in the group.

3) Why use 2?: Setting scale factors by dividing the maximum scale factor with the power of 2 has two key advantages. First, we can guarantee the lower bound of the quantization level. When a channel is assigned to a quantization group, the absolute maximum of the channel is at least larger than half of the threshold of the group. Thus, even for the worst case, $n-1$ bits are utilized for n-bit quantization. Second, when the ratio between the scale factors is a power of 2, we can efficiently compute the matrix multiplication involving decomposed quantized activation tensors with a negligible latency overhead, which we discuss in the following paragraph. In summary, the benefit of the "power of 2" is that it not only enables rescaling with integer arithmetic but also considers the alignment with the channel distribution in activations.

Runtime Requantization. Although tensor decomposition can effectively reduce the quantization error, naively employing it requires an additional computation step to retain functionality (Figure 5(a)). In a naive implementation, we must decompose channels to generate multiple separate matrices and compute the partial product of each group. Each partial product is dequantized using the scale factor of each matrix and added up to the final result. This incurs an additional overhead due to the increased number of floating-point operations. Furthermore, decomposing the channels and computing the partial product of each channel group lead to a shortened reduction axis. This results in underutilization of compute cores especially in the systolic array architecture. To alleviate these inefficiencies and fully utilize the compute cores, we propose *Runtime Requantization*.

Our intuition is that the systolic array accumulator has a sufficiently large bit width, and we can safely *requantize* the partial products with a proper rescaling factor without clipping values due to limited bit width. In this way, we *requantize* accumulated partial products without involving explicit floating-point operations. We use a shifter residing next to the accumulated sum for requantization since we use 2 as a rescaling factor. Figure 5(b) illustrates our runtime requantization approach. We perform matrix multiplication with a channel group that has the larger scale factor first. Then, before computing matrix multiplication of the next group, Tender shifts the accumulated integer value by 1-bit. After finishing the matrix multiplication of all the groups, we dequantize the final result using the smallest scale factor. This approach incurs a negligible latency overhead for rescaling with a small hardware extension. To reduce the overhead of determining a scale factor at runtime, we further optimize our scheme to use calibration to pre-compute the scale factors and biases of each channel offline. We also classify each channel into a group at calibration time and only apply the metadata to perform quantized matrix multiplications at runtime.

A Walking Example. We explain how our decomposed quantization algorithm works with an example in Figure 4, where there are six channels (channel IDs 1-6). After subtracting the bias, each point represents the absolute maximum value of each channel (CMax). In the example, the 2nd channel has the largest CMax value among the channels. Thus, we set the first (and largest) scale factor (`S1`) as the absolute maximum value of the 2nd channel (i.e., 22.4) divided by $k = 2^{b-1} - 1$ (e.g., 127 and 7 for INT8 and INT4 quantization), where b is the quantization bit width. We then set the subsequent quantization boundaries by dividing `S1` by power of 2. For simplicity, we only consider three groups in the example, so `S2` and `S3` would be 11.2 and 5.6 divided by $k = 2^{b-1} - 1$, respectively. Now, we classify each channel into one of the three groups. In the example, the 1st, 3rd, and 5th channels are assigned to the same group (`A3`), sharing the same scale factor (`S3`). The 4th and 6th channels are also classified into the same group (`A2`). The 2nd channel is assigned to another group (`A1`) with the largest scale factor (`S1`). After classification, six different channels are decomposed, so we can minimize the quantization error while making the runtime requantization viable.

Optimization. We apply the Tender algorithm to large language models and achieve similar or better model performance compared to the existing quantization works [20], [21], [65] on INT8 quantization. We achieve this while keeping the algorithm *hardware-friendly* and the required hardware *minimal*. However, for INT4 quantization, the proposed algorithm inevitably leads to some drops in model performance compared to FP16 due to its simplicity, most of which can be easily recovered with conventional row-chunking techniques.

For example, we can divide the rows of the activation tensor into several chunks and calibrate the bias and scale factor for each row chunk. Figure 2 shows that there exist not only inter-channel variances but also *intra-channel* variances. Thus, we may consider intra-channel variances as well as inter-channel variances to group values more effectively. We observe that

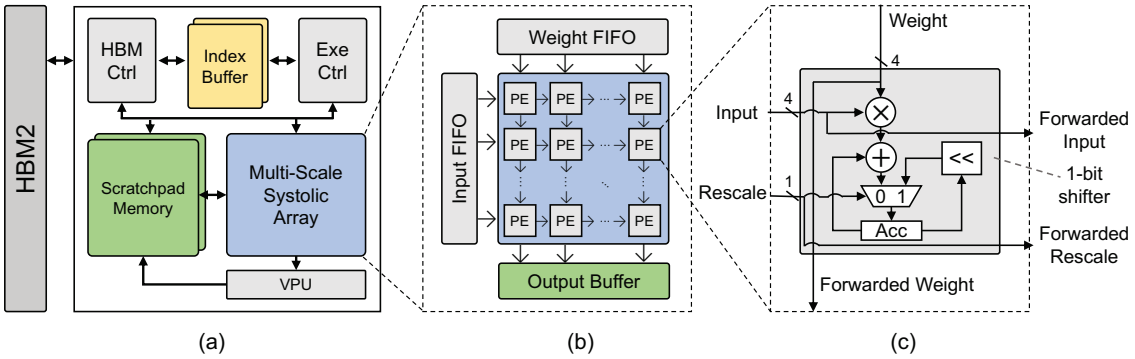


Figure 6: (a) Overview of Tender architecture. (b) Multi-Scale Systolic Array with FIFOs attached for skewing data. (c) Each PE is extended with a 1-bit shifter and a 1-bit control signal for rescaling. The PE updates an accumulated partial sum depending on the rescale signal.

this can greatly improve the accuracy of our algorithm by taking the characteristics of each row chunk into account. Because matrix multiplication is typically performed in a tiled manner (i.e., tile-by-tile), row chunking naturally fits the execution model with almost no additional complexity. We use 256 as a row chunk size. We also apply per-column weight quantization and per-head activation quantization, which also incur negligible calibration overhead.

When optimizing with row chunking, channel grouping is applied to each row chunk, and channel indices, biases, and scale factors are calibrated *independently* for each chunk offline. This makes the group size and compute ordering differ *between* row chunks. From the hardware perspective, the row chunk size needs to be larger than the systolic array dimension; otherwise, there can be underutilization of the compute unit. This is because the systolic array computes rows and applies runtime requantization at the granularity of the systolic array dimension. While the on-chip buffer reuse decreases, it has a negligible impact on performance with a reasonably large chunk size. As mentioned, we choose 256 as a balance point, where accuracy remains close to the baseline model due to fine-grained row grouping, and it is also sufficiently larger than the systolic array dimension.

IV. Hardware Architecture

We quantize the weights and activations in a Transformer block into INT4/INT8, adopting the systolic array as the main computation module. The Tender architecture closely follows the conventional systolic design [28], [29] with a simpler hardware configuration for brevity. In this section, we address how our proposed channel decomposition and runtime requantization are implemented in hardware with a minimal extension to the existing design.

A. Overview

Figure 6(a) shows an overview of the Tender architecture. Memories are HBM2, Scratchpad Memory, Output Buffer, and Index Buffer. The HBM Controller manages data movement between the on-chip buffers and HBM2. The Execution Controller sends an address to the Scratchpad Memory to bring data from the on-chip memory to the systolic array for computation. It also sends control signals to the systolic array to manage its operations. The data is computed in the Multi-Scale Systolic Array (MSA) and Vector Processing Unit (VPU). Figures 6(b) and 6(c) show our Multi-Scale Systolic Array architecture, which follows conventional architectures with minimal extensions of a 1-bit rescale signal and a 1-bit shifter inside the processing element (PE).

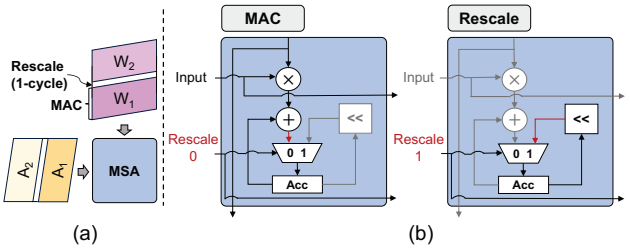


Figure 7: Execution model of the MSA. (a) A 1-cycle bubble is inserted between the decomposed matrices. (b) Activated datapath inside PE during MAC and Rescale operation.

B. Multi-Scale Systolic Array (MSA)

The main matrix multiplication operation occurs in our Multi-Scale Systolic Array (MSA). As illustrated in Figure 6(b), the MSA is a 2-D mesh of PEs with FIFOs attached for skewing the inputs and weights. In Tender, we use a single 64×64 systolic array with each PE executing a 4-bit MAC operation per cycle. When the model precision is INT8, 4 PEs are grouped to perform 8-bit multiplication, with each PE handling either upper or lower 4 bits of inputs and weights. Note that we can scale the number of systolic arrays similarly to the commercial design.

MSA is an output stationary compute module where a partial sum is accumulated in each PE, with an extension of the 1-bit shifter attached to the accumulator register. Figure 7 shows how our MSA works on a series of decomposed matrices with different scale factors. During the computation of each decomposed matrix, PEs perform normal MAC operations, and the accumulated value is stored in the internal 32-bit register (Acc). When the PEs complete matrix reduction for a decomposed matrix (e.g., $A_1 \times W_1$), a 1-cycle bubble is inserted for the input and weight with a rescale signal set (Figure 7(a)). During the rescale operation, it requantizes the

values by shifting 1 bit of the accumulator register to the left. Since each PE finishes matrix reduction for a given decomposed matrix at different cycles, the rescale signal is synchronized with the wavefront of the input and is passed along to the next PE in the same row. To generate rescale signals at the right cycles, the Execution Controller has the metadata of the number of channels executed, indices of tensor splitting points, and rescale factors (if needed).

We choose our systolic array to be output stationary for two reasons. First, it allows us to easily extend the systolic array to handle arbitrary rescale factors (i.e., other than $\alpha = 2$) when needed. Since the accumulator register is wired with the multiplier and shifter within a PE, we can split the register into equal parts and multiply each part with an arbitrary integer rescale factor that can come from the 4-bit input or weight datapath. For example, with a given rescale factor α (e.g., 3) coming from the input datapath, the accumulator is split into 8 parts (each with 4 bits). Then, at each cycle, it is multiplied with the rescale factor, starting from the part with the lowest bits, and its partial product is shifted to the original position to be added to the resulting value. The process is repeated for eight cycles to compute all the parts for rescaling.

Second, the dataflow of the output stationary systolic array requires a minimal hardware extension for rescaling. The output stationary systolic array can be seen as an output value mapped to each PE, and a partial product is produced and accumulated in the same PE every cycle. Thus, an additional 1-bit shifter with control logic for each PE is enough for conventional hardware. For weight stationary design, we need a shifter in the accumulator (which resides outside of PE arrays) as well as in the PEs. Rescaling can be done as follows: 1) Weights are loaded in the group order, and the PEs at the boundary of each group are programmed to shift the partial product after MAC operations. 2) Each corresponding accumulator shifts its value before adding an incoming partial product. Note that although the above procedure requires slightly more changes in hardware than output stationary, it is still a small extension to existing hardware, and Tender can also be implemented on the weight stationary design.

C. Vector Processing Unit (VPU)

The Vector Processing Unit (VPU) is a SIMD-style floating-point unit (FPU) that operates on vector elements. It performs scaling of incoming INT32 results from the Output Buffer (i.e., matrix multiplication results) into INT4/INT8 with an optional activation (e.g., ReLU, GeLU) before storing it back to the Scratchpad Memory. It uses calibrated bias and scale factors, which are computed before inference. VPU consists of 64 FPUs and internal vector registers for pipelining. There are additional registers to buffer scaling factors for quantization. Note that VPU also performs computation for the softmax and LayerNorms in the Transformer block.

D. Controllers & Index Buffer

The Execution Controller and HBM Controller operate independently during computation to keep the MSA busy. The

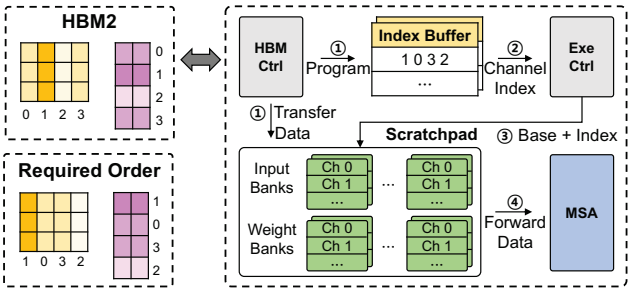


Figure 8: Dataflow of Tender.

HBM Controller handles data transfers between HBM2 and Scratchpad Memory, where the weights, inputs, and computed outputs are stored at INT4/INT8 precision. The Execution Controller sends an address to Scratchpad Memory and control signals (e.g., enable, rescale, and done signal) to the MSA.

As discussed in Section III-B, Tender needs certain channels to be processed before the others (e.g., channel order 1-0-3-2 in Figure 8). To avoid explicit reordering of the data layout in memory, which incurs costly read and write operations, we instead *implicitly* reorder channels through indirect indexing. Figure 8 shows how channels are sent to the MSA in the required order. Specifically, we first store the computation order of the channel indices in the Index Buffer, which is pre-determined at static time through calibration (① Program). The channel indices are reused over the entire row group to amortize memory access overhead, and the Index Buffer is double-buffered to hide memory access latency as it is on the critical path. While the HBM Controller also sends data from the off-chip memory into the Scratchpad Memory (① Transfer Data), the Execution Controller looks up in the Index Buffer and obtains the channel indices (②) to generate an address for the target channel to load (③). Finally, channels are sent to the MSA in the order of the required computation (④).

E. Scratchpad Memory & Output Buffer

As previously mentioned, all the inputs and weights are quantized into INT4/INT8 and stored in the Scratchpad Memory. Without the need for mixed precision storing, the memory access is aligned, and the addressing logic becomes simpler. The Output Buffer stores the computation result from the MSA in INT32 and sends them to the VPU for rescaling back to INT4/INT8, which is also highly banked to match the compute throughput of the VPU.

V. Evaluation

A. Experimental Methodology

Software Implementation. We implement our algorithm using PyTorch Hugging Face [63]. For evaluation, we use the Open Pre-trained Transformers (OPT) suite [72], LLaMA [57], and Llama-2 [58] with varying model sizes ranging from 6.7B to 70B to demonstrate the general applicability of our proposed method. We mainly evaluate language modeling tasks using WikiText-2 [37] and Penn Treebank (PTB) [36] datasets. We use perplexity as an evaluation metric, which is a widely used

TABLE II: INT8/INT4 PTQ results (perplexity) for large language models. Lower is better. We omit LLaMA-65B due to the unacceptable increase in perplexity in all other schemes except for Tender in INT4 quantization.

Precision	Scheme	OPT-6.7B		OPT-13B		OPT-66B		Llama-2-7B		Llama-2-13B		Llama-2-70B		LLaMA-7B		LLaMA-13B	
		Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB	Wiki	PTB
FP16	Base	10.86	13.09	10.13	12.34	9.34	11.36	5.47	20.83	4.88	28.93	3.32	14.44	5.68	8.80	5.09	8.07
INT8	SmoothQuant	**10.93**	13.21	10.40	12.53	9.87	11.71	48.54	1E+4	447.52	491.51	17.30	46.96	27.85	54.98	16.02	32.84
	ANT	19.72	27.96	4E+3	3E+3	3E+3	3E+3	8.79	4E+4	20.52	152.01	7.28	36.18	8.52	13.41	7.49	10.85
	OliVe	**10.93**	13.23	10.28	12.41	**9.43**	11.41	8.16	30.12	30.50	26.16	50.94	245.09	53.34	113.48	7.62	10.76
	Tender	**10.93**	**13.14**	**10.17**	**12.39**	**9.43**	**11.40**	**5.77**	**18.95**	**5.09**	**21.13**	**3.48**	**14.23**	**5.87**	**9.05**	**5.28**	**8.27**
INT4	SmoothQuant	5E+4	2E+4	9E+3	1E+4	6E+4	3E+4	3E+5	3E+5	4E+4	4E+4	7E+4	5E+4	3E+5	2E+5	2E+5	2E+5
	ANT	9E+3	6E+3	4E+4	3E+4	1E+4	7E+3	189.72	2E+4	165.19	1E+3	24.96	155.92	80.13	109.21	96.71	247.65
	OliVe	50.83	43.96	35.76	75.37	6E+3	4E+3	44.24	860.93	1E+3	**97.93**	99.91	216.53	195.15	359.43	94.32	181.69
	Tender	**13.56**	**16.28**	**16.43**	**19.92**	**12.38**	**14.01**	**36.47**	**114.44**	**55.08**	208.76	**13.43**	**50.66**	**23.85**	**38.09**	**13.68**	**28.24**

one for autoregressive models; lower perplexity means better model performance. To show that our algorithm works on the encoder-only model as well, we also evaluate the accuracy of BERT-Large [13] with the GLUE benchmark [60]. We use 128 samples from the Pile [17] validation set for calibration to set scale factors, group indices, and a bias before runtime.

Quantization Baselines. We compare the model performance of our quantization scheme with a variety of outlier-aware PTQ works. For software-only quantization work, we compare with SmoothQuant [65]. SmoothQuant migrates the quantization difficulty of activations to weights by scaling channels of inputs and weights.[1] We also compare our work with ANT [21] and OliVe [20], which target quantization under architectural support. OliVe employs outlier-victim pair encoding, which sacrifices the normal value next to the outlier to preserve the important outlier value. ANT proposes to adaptively use different datatypes for different tensors. They both use custom data formats.

Hardware Implementation. We implement Tender in RTL with SystemVerilog and verify the functionality of each component via RTL simulation. We report the area and power of Tender by synthesizing the components using a commercial 28 nm technology node with Synopsys Design Compiler [53]. On-chip SRAMs are also synthesized from a commercial memory compiler with the same technology. HBM2 [4] is used as off-chip memory with the energy model from FG-DRAM [40]. We also implement a cycle-level simulator with Ramulator [31] for DRAM timing to compare the performance of Tender and baseline accelerators. The timing parameters of the simulator are set based on the RTL synthesis results. Section V-C discusses the detailed configuration of Tender with the performance reported from our simulator.

Accelerator Baselines. We compare the performance and energy efficiency between Tender and existing quantization-based hardware accelerators:

- OLAccel [44] proposes outlier-aware quantization, which represents normal values in 4 bits and outliers in 8 or 16 bits. In addition to the 4-bit normal PEs, OLAccel implements outlier PEs; outlier PEs perform mixed precision computation (e.g., 16-bit $\times$ 4-bit).
- ANT [21] implements a systolic array with a decoder attached to the edge of the array to support various formats including custom datatypes. The decoder converts datatypes into the exponent and integer. We implement an output stationary systolic array as it shows the best performance.
- OliVe [20] also implements an output stationary systolic array with decoder logic to decode datatypes including outlier-victim pairs into the exponent and integer.

For an iso-area comparison, we synthesize the MAC units and accumulators of each accelerator and configure the number of PEs accordingly. We extend the baseline accelerators to use a 32-bit accumulator due to the large reduction length of matrix multiplications in LLMs. Also, we set the same memory bandwidth and on-chip buffer size for the accelerators, which are large enough to fully utilize the compute core. We compare speedups in LLMs with a batch size of 1. The input to output sequence length is set to 2048:1, following the speedup evaluation in prior works [20], [65]. For the generation stage, Tender still works and provides benefits by decomposing the activation. However, the under-utilization issue of most commercial accelerators (e.g., GPU, TPU) can be large as prior work points out [24]. To mitigate this, there are ongoing studies on batching decoding [51], [68], and Tender can work synergistically with those schemes.

B. Language Model Performance

PTQ Performance on LLMs. We analyze the perplexity of PTQ for LLMs, which is the main target of our work. Table II shows the perplexity under INT8 and INT4 quantization settings. The sequence length is set to 2048. For a fair comparison with prior works, we disable the quantization of Tender for matrix multiplication between activations. In INT8 quantization, Tender consistently retains almost the same perplexity from that of the FP16 baseline (less than a 6% increase), while prior works show up to a $1893\times$ increase in perplexity. Notably, Tender even outperforms the FP16 baseline in the Llama-2 models with the PTB dataset. This can be due to the rounding in the quantization function.

[1]We use the original implementation of SmoothQuant. The increase in model performance by enhanced SmoothQuant [25] was marginal while taking far longer calibration time in our experiments. The model performance of enhanced SmoothQuant was still *worse* than Tender.

TABLE III: INT8/INT4 PTQ results (perplexity) across different sequence lengths. Lower is better.

Precision	Scheme	2048		256		32	
		Wiki	PTB	Wiki	PTB	Wiki	PTB
FP16	Base	10.86	13.09	19.18	22.00	78.97	103.42
INT8	SmoothQuant	**10.93**	13.21	**19.17**	22.14	79.32	**102.68**
	ANT	19.72	27.96	48.43	57.97	396.01	364.00
	OliVe	**10.93**	13.23	19.24	22.29	79.69	104.42
	Tender (all)	10.98	13.19	19.31	22.08	78.93	102.99
	Tender	**10.93**	**13.14**	19.28	**22.06**	**78.81**	102.84
INT4	SmoothQuant	5E+4	2E+4	5E+4	2E+4	4E+4	2E+4
	ANT	9E+3	6E+3	8E+3	6E+3	6E+3	3E+3
	OliVe	50.83	43.96	88.05	113.53	441.03	371.73
	Tender (all)	17.15	23.25	27.57	30.58	96.34	118.85
	Tender	**13.56**	**16.28**	**23.16**	**26.12**	**91.27**	**111.90**

Since the overall quantization error is quite low in INT8, rounding can prune out the unnecessary small values, so that the model can instead focus on the important ones. In INT4 quantization, the outlier affects the model performance more profoundly than in INT8 quantization. This is because quantizing outliers with others leads to larger scale factors, and this effect becomes more pronounced in INT4 PTQ which inherently has small quantization levels. Thus, isolating the outlier channel is more important in INT4 quantization. Tender shows far better perplexity than others, which indicates that the channel decomposition of Tender can well separate the outlier channels from others and classify the channels with similar ranges into the same group.

Sequence Length Sensitivity. Table III shows the perplexity comparison between Tender and prior works for three different sequence lengths (2048, 256, 32) on OPT-6.7B [72]. Here we configure Tender into two variants. "Tender" disables the quantization for matrix multiplication between activations for a fair comparison, while "Tender (all)" quantizes all the matrix multiplications in the Transformer block. Tender shows the best model performance for most of the quantization scenarios. Notably, although Tender (all) shows a slight increase in perplexity, it even outperforms the prior works that do not quantize matrix multiplication between activations in most cases. Furthermore, Tender maintains the perplexity close to the FP16 baseline even when the sequence length increases. This is due to the channel decomposition which considers inter-channel variation and the row chunking which handles intra-channel variation. As shown in the results, Tender is more robust than others while dealing with diverse scenarios of outlier values and sequence lengths. Note that we use single calibration data attained from the 2048 sequence length for the evaluation across different sequence lengths.

Quantization Accuracy on BERT. Table IV shows the accuracy for INT8 and INT4 quantization on BERT-Large [13] with the GLUE benchmark [60]. All schemes in Table IV quantize *all* the matrix multiplications in a Transformer block. Although the outliers of the BERT-Large are much smaller than the ones of other large language models, Tender outperforms other baselines in many tasks. This indicates that our algorithm also benefits encoder-only and relatively small models.

TABLE IV: INT8/INT4 PTQ results (accuracy) on BERT-Large. Higher is better.

Precision	Scheme	CoLA	SST-2	MRPC	STS-B	QQP	QNLI
FP32	Base	60.20	93.12	91.58	89.94	91.40	92.33
INT8	ANT	59.16	92.55	77.99	89.23	89.66	81.48
	OliVe	**61.12**	93.12	91.33	89.91	91.42	92.02
	Tender	60.45	**93.23**	**91.55**	**89.98**	**91.43**	**92.31**
INT4	ANT	53.77	90.60	21.09	85.93	83.62	60.86
	OliVe	59.02	92.09	85.32	87.43	**89.72**	**90.48**
	Tender	**61.78**	**92.32**	**89.42**	**87.77**	89.23	90.29

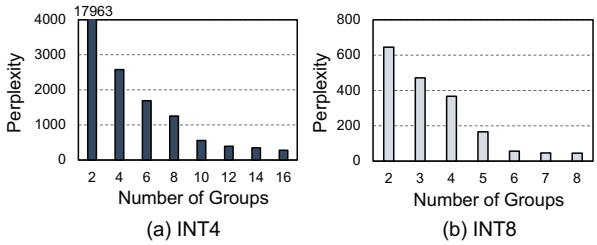


Figure 9: Perplexity for the different number of groups in (a) INT4 and (b) INT8 quantization. Lower is better.

Multi-Scale Quantization. Figure 9 shows the perplexity on Llama-2-7B [58] while varying the number of groups for channel decomposition. We use the PTB [36] dataset and a fixed sequence length of 256. As we increase the number of groups, the perplexity decreases rapidly for both INT4 and INT8 quantization. This shows that separating the channels only into two groups (i.e., outlier channels and normal channels) is not enough, and decomposing the channels into *multiple* groups is necessary to achieve better model performance. Note that naively adopting multi-scale quantization results in a frequent interrupt during matrix multiplication. However, Tender handles the multi-scale quantization without interrupts by exploiting a minimally extended systolic array.

C. Tender Performance

Area and Power. Table V shows the architectural configurations of Tender. Functioning at the 1 GHz clock frequency, Tender has an area of 3.98 mm^2 with the peak power consumption of 1.60 W. The numbers in the systolic array are MAC units and 32-bit accumulators combined. To match the compute throughput of the VPU, we also design the output buffer to be highly banked while trading off area with throughput. We configure the PEs of the baseline accelerators to have the same area and clock frequency as the ones in Tender for performance and energy evaluation.

Performance. Figure 10 shows the speedup of Tender and other accelerators over ANT for the LLM models in Section V-B; for brevity, we omit LLaMA as it shows the results similar to Llama-2. The source of speedups mainly comes from the careful algorithm-hardware co-design of Tender. Using single precision of INT4 with a hardware-friendly

TABLE V: Area and power characteristics of Tender.

Component	Setup	Area [mm^2]	Power [W]
Systolic Array	64×64 PEs	2.00	1.09
Vector Processing Unit	64 FPUs	0.08	0.02
Input/Weight FIFOs	64×2	0.05	0.34
Index Buffer	2×(16KB)	0.23	0.01
Scratchpad Memory	2×(256KB)	1.15	0.13
Output Buffer	64KB	0.47	0.01
Total		**3.98**	**1.60**

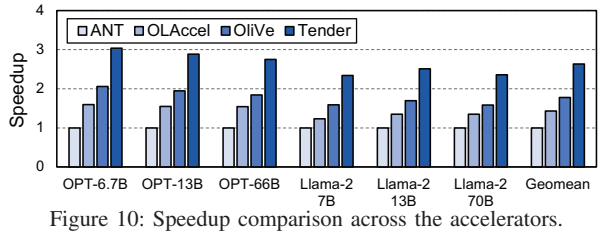


Figure 10: Speedup comparison across the accelerators.

tensor decomposition scheme, Tender enables a simpler and denser systolic array design with higher throughputs compared to others. In contrast, OLAccel has complex control logic and outlier PEs to support mixed precision, and ANT and OliVe shift the multiplication result of the integers with the exponent sum and require more hardware resources. Tender only performs INT4 MAC operations and eliminates the need to handle higher precision numbers. Tender shows higher speedups compared to OliVe since OliVe computes using the exponent and integer. ANT performs worse than other accelerators because most of the layers use 8-bit precision to compensate for the quantization loss. Overall, Tender achieves 2.63×, 1.84×, and 1.48× speedups over ANT, OLAccel, and OliVe with better model performance and minimal extensions to MAC units.

Energy Efficiency. Figure 11 shows the energy efficiency of Tender and the baseline accelerators under the same off-chip memory and on-chip buffer size. The energy efficiency of Tender mainly comes from a smaller memory size with efficient hardware computation under the INT4 precision. Compared with OLAccel, the energy efficiency comes from FIFO registers, off-chip memory access, and compute units due to the shorter computation time. For ANT, using mixed precision incurs more off-chip accesses and longer computation latency, leading to higher energy consumption than Tender and OLAccel. Compared to OliVe, Tender shows better efficiency due to the denser systolic array design. Overall, Tender shows 1.84×, 1.53×, and 1.24× higher energy efficiency than ANT, OLAccel, and OliVe.

VI. Analysis and Discussion

A. GPU Implementation of Tender Decomposition

Tender uses standard INT4 and INT8 representations, thereby enabling straightforward deployment to existing systems. To demonstrate the non-intrusive characteristics of Tender, we implement our quantization algorithm on NVIDIA

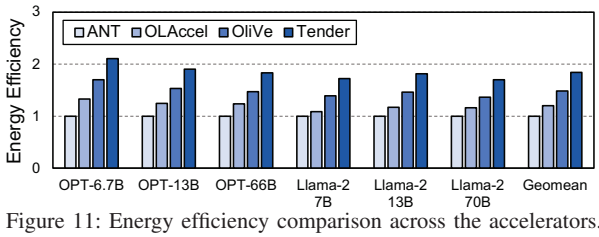


Figure 11: Energy efficiency comparison across the accelerators.

GPUs. Figure 12 shows the normalized latency of Tender software and other quantization schemes in Section II-C. All schemes are implemented with CUTLASS INT8 GEMM kernel. We use OPT-6.7B for RTX 3090 and OPT-66B for A100 to saturate compute units. For small models on A100, we observe that per-tensor INT8 GEMM exhibits similar latency to FP16 due to compute underutilization and the relatively close tensor core throughput between INT8 and FP16. Latency and mean square error (MSE) are measured for each scheme with a sample from the query projection in Layer 16.

As shown in the results, Tender SW shows an MSE similar to the "per-channel" approach and provides slight performance benefits over FP16. However, it does not realize its full potential (i.e., similar to "per-tensor/-row") due to the need of explicit dequantization on GPUs. The overall GEMM execution also takes longer due to the repetitive operations on smaller submatrices. In addition, INT GEMM kernels for tensor cores require 128-bit aligned memory access, which necessitates the padding of each subtensor (to the multiple of 16) before computation. Accelerators that support Tender can avoid the overheads and help achieve the full potential of Tender.

B. Accelerators with Floating-Point Arithmetic

Variants of FP8 formats for DNN training/inference show good model performance [38]. Although using low-bit FP

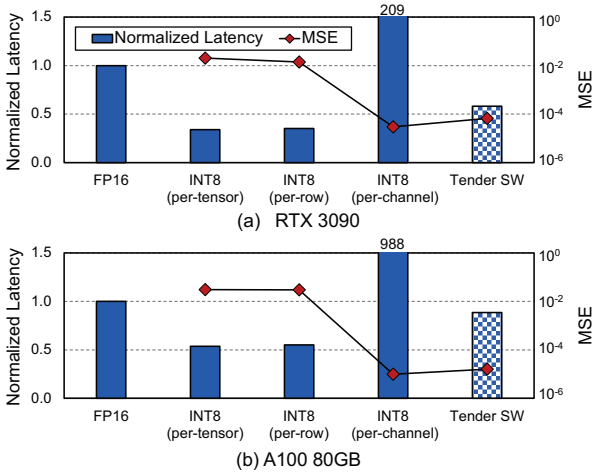


Figure 12: Comparison of Tender SW and other schemes on GPUs. Latencies are measured in (a) RTX 3090 and (b) A100 80GB.

TABLE VI: PTQ perplexity of Tender and MSFP for WikiText-2.

Precision	OPT-66B	Llama-2-70B	LLaMA-65B
FP16	9.34	3.32	3.56
MSFP12	7E+3	74.61	73.22
MSFP12-OL	56.69	15.57	26.11
Tender-INT4	**13.38**	**13.43**	**9.30**

TABLE VII: Accuracy for the lm-evaluation-harness zero-shot tasks used in [10], [48]. Higher is better. Tender uses INT4.

Tasks	OPT-6.7B				LLaMA-7B			
	FP32	SMX4	MXFP4	Tender	FP32	SMX4	MXFP4	Tender
Hellaswag	67.16	26.94	54.13	**64.54**	76.20	25.89	**67.51**	57.30
WIC	48.12	49.84	**51.72**	50.00	49.06	**50.00**	46.24	49.53
Anli-r2	34.40	33.40	33.90	**34.20**	36.10	33.40	**35.30**	35.20
Winogrande	65.43	50.12	52.88	**61.80**	70.01	50.59	**62.35**	59.04
ARC easy	60.02	29.76	44.57	**56.82**	72.85	27.78	**63.68**	58.50
ARC challenge	34.73	23.46	29.18	**33.79**	44.71	26.88	35.49	**36.26**
Lambada	67.69	00.02	43.74	**60.06**	73.61	00.02	56.65	**56.80**
College CS	34.00	25.00	25.00	**34.00**	26.00	23.00	22.00	**28.00**
Int. law	37.19	23.97	**32.23**	26.45	46.28	29.75	33.06	**33.88**
Jurisprudence	21.30	**25.93**	25.00	21.30	36.11	**26.85**	**26.85**	24.07

hardware could reduce the impact of outliers, it is more area/energy *inefficient* than integer compute units [10]. MSFP [9] uses a shared exponent to mitigate the inefficiency. As shown in Table VI, however, Tender shows better performance than MSFP. By default, MSFP12 uses an 8-bit shared exponent for 16 elements in a *row*, so the huge increase in perplexity likely comes from sharing exponents between outliers and others. We modify the MSFP12 to use the shared exponent for 8 elements in a column (MSFP12-OL). However, still Tender shows better model performance. This is likely because the intra-channel variance of the outlier channel is more precisely represented in the integer format than the MSFP. Thus, Tender achieves better performance than MSFP, while requiring much simpler hardware.

C. Comparison with Microscaling (MX) Formats

Shared Microexponents (SMX) [10] and the subsequent Microscaling (MX) format [41] are recently proposed number representations that employ multiple levels of scaling.[2] Similar to MSFP, they group elements in a block-based manner, but with two-level scaling, where the scaling factors are constrained to powers of two. SMX groups 16 elements with an 8-bit shared exponent, and two elements in a block form a subgroup to share a 1-bit subscale factor. Similarly, MX groups 32 elements, but each element has also its *own* exponent field in addition to the 8-bit shared exponent.

Table VII compares the accuracy between employing Tender and using SMX and MX formats on OPT-6.7B and LLaMA-7B for the lm-evaluation-harness tasks used in [10], [48]. For a fair comparison, we employ the same compute flow across the low-precision formats while quantizing matrix multiplications into low precision and keeping other element-wise operations as scalar floating-point formats as in [48]. The results show that Tender can provide better or comparable accuracy while it builds on standard INT4 representations. Note that SMX and MX need more customized compute units. For instance, each element in an MXFP block is essentially a floating-point number, requiring hardware that deals with FP computation.

The essence of Tender involves setting scale factors that are powers of two *apart* between the groups (e.g., the *ratios* of scale factors between $1^{st}/2^{nd}/3^{rd}$ immediate neighbor groups: $2^1, 2^2, 2^3$), which enables implicit rescaling with minimal overhead (1-cycle). MX formats, however, merely use some power-of-two scale factors, similar to MSFP. Thus, implicit rescaling by 1-bit shifting cannot be achieved by simply using MX. As discussed in Section III, the scale factor of Tender is also *not* limited to powers of two; it can be any real number. Additionally, MX formats follow a conventional approach of grouping adjacent elements, whereas Tender groups columns within similar ranges while considering ease of computation.

D. Tender on Output and Weight Stationary Dataflows

While we present the benefit of Tender based on the systolic array that employs the output stationary dataflow, it is not a strict requirement as discussed in Section IV-B. For the generation stage in LLMs, we may consider batching inputs only up to the number of rows of an output stationary systolic array due to compute and energy efficiency, while we can batch more inputs than the systolic array dimension for the weight stationary design. If there are ample batching opportunities, weight stationary can be more efficient since the output stationary design may incur idle time and additional energy due to repeated weight loading. Conversely, when batching is limited, such as by the memory size of large intermediate states (i.e., key-value cache), output stationary could be as efficient as weight stationary since it minimizes the movement of high-precision partial sums. Note that Tender can be employed in either case and provides benefits.

E. Channel Decomposition and Compute Utilization

Although channels are decomposed into multiple groups, Tender can continuously compute the groups in MSA without a major interrupt. This is because rescaling can be done entirely in integer PEs (implicit) and only takes a single cycle (i.e., runtime requantization). During runtime, skewed channel groups are *continuously* provided with a 1-cycle rescale signal, aligned with the inputs between the groups (Figure 7(a)), and each PE individually rescales the accumulated value via 1-bit shifting; this feature is crucial as the systolic array takes as input a skewed matrix. The original computation in Figure 5(a) leads to large under-utilization of compute units. Tender does not suffer from this issue and *preserves* the reduction axis of the original input matrix, regardless of the number of groups or the group size for the matrix. Thus, during offline calibration, Tender only considers model performance to determine the number of groups, where perplexity does not decrease further as shown in Figure 9; it varies at some point due to noise. Each channel is then classified into the corresponding group. This calibration process naturally determines the size of each channel group.

[2]In this paper, we denote Shared Microexponents [10] as SMX to distinguish it from the Microscaling (MX) formats [41] endorsed by OCP.

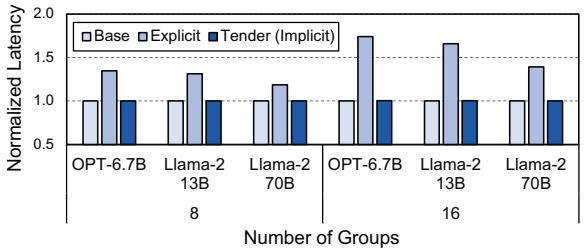


Figure 13: Comparison between implicit and explicit requantization.

F. Impact of Implicit and Explicit Requantization

Channel decomposition with *explicit* requantization leads to lower compute utilization due to the shortened reduction axis and also increases the number of FP operations. To understand the benefit of implicit requantization on Tender hardware, Figure 13 presents the end-to-end execution time when Tender employs either implicit or explicit requantization, which is normalized to per-tensor quantization (Base). Note that larger models (e.g., Llama-2-70B) generally need more number of groups to attain reasonable accuracy. The results show that Tender with explicit requantization greatly degrades performance, by up to a 1.74× slowdown over the baseline. Also, as the larger number of groups (e.g., 16) further reduces the reduction axis, we observe larger slowdowns compared to ones in the smaller number of groups. On the other hand, Tender (Implicit) offers almost the same execution time as the baseline. This is because there is only a 1-cycle requantization overhead for each group, so increasing the number of groups barely affects performance. By employing implicit requantization, Tender effectively minimizes the overhead associated with channel decomposition.

VII. Related Work

DNN Accelerators. Domain-specific accelerators for DNNs have been extensively studied over the past decade [3], [7], [8], [14], [15], [23], [24], [28], [39], [45], [47]. The processing units and dataflow of these accelerators are highly specialized for DNN computation, leading to high performance and energy efficiency. Several accelerators adopt near-memory processing to overcome the memory-bound characteristics of specific types of DNN workloads [6], [30], [32], [34]. Other works also target sparsity in DNNs to skip ineffective computation [2], [22], [35], [43], [46], [61], [67], [71]. Tender is orthogonal to these works and can be synergistically used with conventional systolic array-based DNN accelerators.

DNN Quantization. Quantization-aware training (QAT) trains the model under quantization to make it adapt to quantization errors [26], [54]. However, QAT is a limited option due to the large model sizes, and thus post-training quantization (PTQ) has been widely studied for LLMs [11], [65], [66]. RPTQ [69] employs K-means clustering to group activation channels and applies asymmetric quantization at the granularity of a channel group. However, each channel group needs to be computed one by one, leading to lower compute utilization due to the smaller matrix sizes of each group. In addition, all the partial products from each group need to be explicitly dequantized to add up and obtain the final resulting matrix, which is costly. GPTQ (OPTQ) [16], AWQ [33], and QLoRA [12] are the recent weight quantization works. GPTQ quantizes weights column by column using the Hessian matrix to compensate the errors. AWQ scales weight channels by observing outliers in activation tensors to reduce quantization errors. QLoRA introduces the 4-bit NormalFloat datatype for block-wise weight quantization by considering the distribution of values.

For quantization under architectural support, BitFusion [49] proposes a bit-flexible architecture that can handle various precisions. BiScaled-DNN [27] introduces a new *fixed-point* (FxP) number format that employs two scale factors to represent values of small and large magnitudes in a tensor. While it offers advantages over conventional FxP, the heuristic for determining scale factors and the nature of its design would not easily be applicable for a larger number of groups, unlike Tender, which is crucial for language model performance. Its element-wise metadata also leads to more intrusive changes in conventional PEs. ANT [21] adaptively uses a variety of datatypes, and OliVe [20] expresses outliers using specialized datatype to reduce quantization errors. However, using custom or mixed datatypes makes it challenging to be deployed for commodity hardware. Compared to these works, Tender offers a non-intrusive but effective solution while building on the conventional number representation and datatype.

VIII. Conclusion

Emerging large language models (LLMs) show remarkable model performance, but quantization for scalability is challenging due to the presence of outliers in the activation tensors. In this paper, we present Tender, an algorithm-hardware co-design that carefully considers both hardware performance and quantization error. Tender minimizes quantization errors by splitting the activation tensors along the feature/channel dimension to separate the outlier channels from the others. We address the runtime overhead of the channel decomposition by introducing implicit requantization with the support of a minimally extended systolic array. Tender significantly improves PTQ performance even for ultra low-bit quantization without mixed-precision computation or custom datatypes. This work opens new possibilities for easier and more efficient deployment of LLMs in a variety of practical scenarios.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by a research grant from Samsung Advanced Institute of Technology (SAIT) and by the artificial intelligence semiconductor support program to nurture the best talents (No. RS-2023-00256081) supervised by Institute for Information & Communications Technology Planning & Evaluation (IITP). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

REFERENCES

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[3] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[4] J. S. S. T. Association, *JEDEC Standard JESD235A: High Bandwidth Memory (HBM) DRAM*, JEDEC, Virginia, USA, 2015.

[5] Y. Bondarenko, M. Nagel, and T. Blankevoort, "Understanding and overcoming the challenges of efficient transformer quantization," in *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[6] D. Chen, H. He, H. Jin, L. Zheng, Y. Huang, X. Shen, and X. Liao, "Metanmp: Leveraging cartesian-like product to accelerate hgnns with near-memory processing," in *ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[7] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[8] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[9] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. Chand Koppaka, X. SONG, S. Som, K. Das, S. T, S. Reinhardt, S. Lanka, E. Chung, and D. Burger, "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[10] B. Darvish Rouhani, R. Zhao, V. Elango, R. Shafipour, M. Hall, M. Mesmakhosroshahi, A. More, L. Melnick, M. Golub, G. Varatkar, L. Shao, G. Kolhe, D. Melts, J. Klar, R. L'Heureux, M. Perry, D. Burger, E. Chung, Z. S. Deng, S. Naghshineh, J. Park, and M. Naumov, "With shared microexponents, a little shifting goes a long way," in *ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[11] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Gpt3.int8(): 8-bit matrix multiplication for transformers at scale," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[12] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[14] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[15] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A configurable cloud-scale dnn processor for real-time ai," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[16] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "OPTQ: Accurate quantization for generative pre-trained transformers," in *International Conference on Learning Representations (ICLR)*, 2023.

[17] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[18] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*, 2022.

[19] Google. (2023) Gemini. [Online]. Available: https://gemini.google.com

[20] C. Guo, J. Tang, W. Hu, J. Leng, C. Zhang, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization," in *ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[21] C. Guo, C. Zhang, J. Leng, Z. Liu, F. Yang, Y. Liu, M. Guo, and Y. Zhu, "Ant: Exploiting adaptive numerical data type for low-bit deep neural network quantization," in *55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[22] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, "Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks," in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[24] S. Hong, S. Moon, J. Kim, S. Lee, M. Kim, D. Lee, and J.-Y. Kim, "DFX: A low-latency multi-fpga appliance for accelerating transformer-based text generation," in *55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[25] Intel. (2020) Intel neural compressor. [Online]. Available: https://intel.github.io/neural-compressor

[26] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[27] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, K. Gopalakrishnan, and L. Chang, "Biscaled-dnn: Quantizing long-tailed datastructures with two scale factors for deep neural networks," in *56th ACM/IEEE Design Automation Conference (DAC)*, 2019.

[28] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[29] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[30] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang, "Recnmp: Accelerating personalized recommendation with near-memory processing," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[31] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters (CAL)*, 2015.

[32] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[33] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," *arXiv*, 2023.

[34] H. Liu, L. Zheng, Y. Huang, C. Liu, X. Ye, J. Yuan, X. Liao, H. Jin, and J. Xue, "Accelerating personalized recommendation with cross-level near-memory processing," in *ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[35] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, "Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture," in *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[36] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, and B. Schasberger, "The penn treebank: Annotating predicate argument structure," in *Proceedings of the Workshop on Human Language Technology*, 1994.

[37] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *International Conference on Learning Representations (ICLR)*, 2017.

[38] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu, "Fp8 formats for deep learning," *arXiv preprint arXiv:2209.05433*, 2022.

[39] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[40] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: Energy-efficient dram for extreme bandwidth systems," in *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[41] *OCP Microscaling Formats (MX) Specification Version 1.0*, Open Compute Project, 9 2023.

[42] OpenAI. (2023) Chatgpt. [Online]. Available: https://openai.com/chatgpt

[43] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[44] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[45] S. Qian Zhang, B. McDanel, and H. T. Kung, "Fast: Dnn training under variable precision block floating point with stochastic rounding," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[46] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[47] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[48] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf, S. Dusan, V. Elango, M. Golub, A. Heinecke, P. James-Roxby, D. Jani, G. Kolhe, M. Langhammer, A. Li, L. Melnick, M. Mesmakhosroshahi, A. Rodriguez, M. Schulte, R. Shafipour, L. Shao, M. Siu, P. Dubey, P. Micikevicius, M. Naumov, C. Verrilli, R. Wittig, D. Burger, and E. Chung, "Microscaling data formats for deep learning," *arXiv preprint arXiv:2310.10537*, 2023.

[49] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[50] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-BERT: hessian based ultra low precision quantization of BERT," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[51] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *International Conference on Machine Learning (ICML)*, 2023.

[52] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "Drq: Dynamic region-based quantization for deep neural network acceleration," in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[53] Synopsys. (2023) Design compiler - synopsys. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[54] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," in *International Conference on Learning Representations (ICLR)*, 2021.

[55] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. Janapa Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference," in *57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[56] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[57] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[58] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[59] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[60] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *International Conference on Learning Representations (ICLR)*, 2019.

[61] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[62] X. Wei, Y. Zhang, X. Zhang, R. Gong, S. Zhang, Q. Zhang, F. Yu, and X. Liu, "Outlier suppression: Pushing the limit of low-bit transformer language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[63] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[64] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[65] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning (ICML)*, 2023.

[66] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, "Zeroquant: Efficient and affordable post-training quantization for large-scale transformers," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[67] A. Yazdanbakhsh, A. Moradifirouzabadi, Z. Li, and M. Kang, "Sparse attention acceleration with synergistic in-memory pruning and on-chip recomputation," in *55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[68] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for {Transformer-Based} generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[69] Z. Yuan, L. Niu, J. Liu, W. Liu, X. Wang, Y. Shang, G. Sun, Q. Wu, J. Wu, and B. Wu, "RPTQ: Reorder-based post-training quantization for large language models," *arXiv preprint arXiv:2304.01089*, 2023.

[70] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[71] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[72] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.